Universidad Rey Juan Carlos

Examen Final Programación Orientada a Objetos. 22 de Diciembre de 2017. **DURACIÓN: 3h**

Estamos interesados en programar una clase Juguete. El juguete tiene los siguientes atributos:

- nombre (obligatorio)
- número de jugadores (opcional)
- edad mínima (obligatorio)
- precio base (es un entero, obligatorio).

Los juguetes pueden ser de carácter general (clase Juguete), pero también pueden ser electrónicos y educativos.

Los juguetes educativos tienen una edad máxima y un atributo "bonificacion en el precio" que se indican al instanciarlos. La edad máxima no puede ser ~~superior~~ inferior o igual a la edad mínima.

Los juguetes electrónicos tienen también un atributo "recargo en el precio" que también se indica al instanciarlo.

Las bonificaciones y recargos han de ser menores que el precio base.

1. **(1.5 puntos)** Programa todos los constructores necesarios en todas clases. Ten en cuenta el carácter obligatorio y opcional de los atributos, así como los valores posibles que cada uno puede tomar. Cualquier error en los valores hará que los constructores lancen una excepción de tipo EinvalidPropertyException, que hay que programar.

2. **(0.5 punto)** Programa un método "getPrecio" que devuelva al precio final aplicando o no recargos o bonificaciones, según el caso.

3. **(1 punto)** Todos los juguetes deben implementar la interfaz "Comparable". Un juguete será menor que otro si su precio (sin contar el recargo o bonificación) es menor. Así mismo, nos ayudaremos de una clase externa para establecer un orden alternativo: un juguete será menor que otro si su precio (contando bonificaciones o recargos) es menor.

4. **(0.5 puntos)** Programad el método "equals" de la clase Juguete. Dos juguetes son iguales si tienen el mismo nombre.

5. **(2.5 puntos)** Programar una clase "Cliente" que tenga un listado de juguetes y un nombre. La clase "Cliente" tendrá los siguientes métodos:

   i. `addJuguete`: añadirá un juguete a su lista. No se pueden añadir dos juguetes iguales.

   ii. `calcularGasto`: recorre todos los juguetes y calcula el precio total que se va a gastar. Recordad que hay juguetes que tienen recargos o bonificaciones.

   iii. `listarJuguetes`: lista todos los juguetes que ha comprado, ordenado por precio base.

   iv. `listarJuguetesByPrecioFinal`: lista todos los juguetes que ha comprado, ordenado por precio final (una vez ha aplicado recargos y bonificaciones).

Se valorará muy favorablemente la elección de la colección/colecciones correctas.

6. **(2 puntos)** Programar una clase "Juguetería" que tenga un listado de clientes. Tendrá tres métodos:

   i. `addCliente`: inserta un cliente. Si el cliente ya existe, se introduce la nueva info y se elimina la anterior.

   ii. `listarJuguetesPorCliente`: recibirá como parámetro el nombre del cliente y retornará el listado de los juguetes que ha comprado. Ojo, que ha de dar excepción si el cliente no existe.

   iii. `listarClientes`: retorna una lista de clientes, ordenado por nombre.

   Importante escoger bien las estructuras más correctas para las colecciones.

7. **(1 puntos)** Programar una clase "Repositorio" que sea capaz de serializar tanto la clase Juguetería como la clase "Cliente". Se valorará muy positivamente que una única clase sea capaz de serializar tanto Jugueterías como Clientes.

8. **(1 puntos)** Probar todos los métodos de todas las clases en una clase Principal. Esta clase no podrá lanzar ninguna excepción. Si se produce una excepción de E/S, se invocará al método printStackTrace. Si se produce otra excepción cualquiera, se escribirá el mensaje por pantalla pero solo llamando al método getMessage() de la excepción. Al finalizar la ejecución sin excepciones, deberá escribir en un archivo de texto llamado "info.txt" el gasto de todos los clientes que se hayan creado. Una fila por cada cliente.

OBSERVACIONES:

- Si, por ejemplo, la lista de juguetes es un HashSet, cualquier método que retorne lista de juguetes bastará con que retorne un HashSet. No es requerido que se hagan conversiones de listas y/o colecciones.

- Ojo con la nomenclatura, **poned nombres descriptivos y usad mayúsculas y minúsculas de manera correcta**.

- Métodos de Map:

Modifier and Type	Method and Description
void	**clear()** Removes all of the mappings from this map (optional operation).
boolean	**containsKey(Object** key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue(Object** value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	**entrySet()** Returns a Set view of the mappings contained in this map.
boolean	**equals(Object** o) Compares the specified object with this map for equality.
V	**get(Object** key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
int	**hashCode()** Returns the hash code value for this map.

boolean	**isEmpty()** Returns true if this map contains no key-value mappings.
Set<K>	**keySet()** Returns a Set view of the keys contained in this map.
V	**put(K key, V value)** Associates the specified value with the specified key in this map (optional operation).
void	**putAll(Map<? extends K,? extends V> m)** Copies all of the mappings from the specified map to this map (optional operation).
V	**remove(Object key)** Removes the mapping for a key from this map if it is present (optional operation).
int	**size()** Returns the number of key-value mappings in this map.
Collection<V>	**values()** Returns a Collection view of the values contained in this map.

- Métodos de Collection:

Methods

Modifier and Type	Method and Description
boolean	**add(E e)** Ensures that this collection contains the specified element (optional operation).
boolean	**addAll(Collection<? extends E> c)** Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear()** Removes all of the elements from this collection (optional operation).
boolean	**contains(Object o)** Returns true if this collection contains the specified element.
boolean	**containsAll(Collection<?> c)** Returns true if this collection contains all of the elements in the specified collection.
boolean	**equals(Object o)** Compares the specified object with this collection for equality.
int	**hashCode()** Returns the hash code value for this collection.
boolean	**isEmpty()** Returns true if this collection contains no elements.
Iterator<E>	**iterator()** Returns an iterator over the elements in this collection.
boolean	**remove(Object o)** Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll(Collection<?> c)** Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	**retainAll(Collection<?> c)** Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	**size()** Returns the number of elements in this collection.
Object[]	**toArray()**

	Returns an array containing all of the elements in this collection. **toArray**(T[] a)
<T> T[]	Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

- A modo de ayuda, se recuerda el nombre de algunas de las clases, métodos y excepciones relacionadas con la E/S: FileWriter, BufferedWriter, PrintWriter, FileOutputStream, ObjectOutputStream, println(), writeObject(), IOException.

- Programad únicamente los get y set que sean necesarios.

- Programad los constructores de las clases "Cliente", "Juguetería".

- Las colecciones en los constructores siempre se inicializan como vacías.